Bot Assisted Software Development

Ujjwel Balwal, Vallidevi Krishnamurthy

Abstract: ***Many software products are direct or indirect implementations of already available algorithms; most of which are Open Source and available online. The problem is that we have to search the vast repositories to find the relevant implementation. We propose a way to catalog the program code, so that a bot could read that catalog to figure out the working of a code block and fetch required functionality that can be customized for the new project. This will increase the productivity of the team, is convenient for the developer community and will greatly reduce initial development time.***

Index Terms: Assisted, Software, Automated, Development

I. INTRODUCTION

Open source communities have a praiseworthy amount of software resources available online. Even for the organizations that do not embrace open source community, they often have a huge amount of beta and final version of software snippets and modules available at their disposal. A survey concluded that about 71% of practitioners believe that software reuse results in better quality [1]. Even though the code is organized, it is still a tedious task to remember and search for the exact module needed for the new endeavor and many times the same functionalities are coded again from scratch, even though some developer had the almost excellent working piece available online. The process of searching the database of modules; if automated, can save a huge amount of time and human resource, also increasing the efficiency and convenience at the same time.

It is practically possible to build a personal assistant, referred to as ‘bot’ throughout this paper that is capable of searching the candidate code snippets from a repository. To create such a system, firstly, we need to catalog the code present in the repository, then we need to create a database based on the catalog information, thereafter we can build an assistant to fetch the required functionality from the created database. Step three, and two are easy to achieve provided we have successfully cataloged the code. In this paper, we propose a concept of cataloging the code without disrupting the current way the code is stored and shared.

The organization of the paper is as follows: Following the Section 1 (Introduction), Section 2 describes the way in which database has to be created for automating the software development process. Section 3 describes the development life cycle of the Bot assisted software development system. Section 4 concludes the paper which is followed by the references and the author information. Appendix A at the end describes the use-case example followed by a Demo script.

Revised Manuscript Received on April 13, 2019.

Ujjwel Balwal, SSN College of Engineering, Anna University, Chennai, Tamil Nadu, India

Vallidevi Krishnamurthy, SSN College of Engineering, Anna University, Chennai, Tamil Nadu, India

II. DATABASE CREATION

The most important part for the bot system is having all the available code correctly cataloged so that the bot can understand what a block of code specifically means. We will achieve this using special comments within the code, without affecting the actual coding part, as described below:

All the programming languages offers the functionality to add comments to the code. Comments can be used to communicate the attributes of the code blocks to specify their behavior and functionality. We name them ‘catalog comments’

It is crucial to separate the comments intended for actual documentation from the comments that are being used for cataloging. This can be achieved using a specified keywords to mark the start and ending of catalog comments, a keyword that should not be common in natural language and is not already a keyword in majority of the programming languages out there. For example, *Foo Comment*.

Thus, a comment starting with *FooComment* will be treated as a catalog comment and will be read by the bot as an attribute. The distinction is given below.

```
// [This marks a documentation comment]
// FooComment [This marks a catalog comment]
```

For providing a block of code with an attribute, the block can be surrounded by an opening marker comment and a closing marker comment, as shown below.

```
/* FooComment Attributes and properties
…
*/

[Your code here (function, class etc.)]

/* end FooComment */
```

The bot should be capable to analyze the code for incomplete comment information like an opening attribute comment but no closing attribute comment and indicate the same as a warning, similar to bracket matching capabilities in programming languages.

Also, we don’t want all the catalog comments to be shipped along with the final product to the customers, so the bot needs to have the functionality to remove all the catalog comments from the code. In this process, the software will search for the opening and closing catalog comment pairs and will remove them to clean the code of any additional information that the developer doesn’t want to go out of the development premises.

We need to come up with a set of attributes that can be used to convey the significance of a code block. The attributes to be included depend on the block of code that is being cataloged. A class will have different attributes as compared to a function. A generic set of attributes that we suggest are provided in the table below.

Take a function as the block that needs to be cataloged; the set of attributes in case of a function being cataloged is specified in Table 1:

Table. 1 Attribute table for a function block

Attribute	Description
parent_id:	foreign key reference to the parent of the function (outer function / class)
id:	equivalent of the primary key in a database
p_lang:	programming language used (not to be included in the comments)
inputs:	input datatype the module takes (logically or syntactically)
syntactic inputs:	inputs that are provided as arguments
logical inputs:	inputs given by input streams, hard-coded inputs
outputs:	logical or explicit output datatype of the final result
i_arg?:	signifies for each input, weather the input is passed as argument (y / n)
o_arg?:	is the output returned as an argument (y / n)
space_complexity:	in bigO
time_complexity:	in bigO
id_keys:	key description, eg: for a text narrator: text, narrator, speech, converter
desc:	long description (if required)
sp_dependencies:	dependencies required (other project files / headers / libraries)

An example follows this for the better understanding of the concepts

```
''' FooComment #function
id: 007
inputs: string[2]
i_arg: y
outputs: string[1]#.csv
o_arg: n
desc: semantic analysis of string
id_keys: sematic, string, csv
parent_id: null
'''

[...The actual code goes here...]

''' end FooComment '''
```

In the above example, we gave the block of code the attributes indicating that the code accepts input as an array of string (inputs: string[2]) in the form of arguments (i_arg=y) and provides the required output as a single string that gets stored into a csv file (outputs=string[1]#.csv) and the csv file is not returned as output return value (o_arg=n).

Now, some of the attributes such as the programming language used and space and time complexity need not be provided in the catalog attribute comments, as the programming language will be known by the bot instinctively because the bot is reading the code. Although, it is always a good practice to include as much information as practically possible, to increase the efficiency of bot's prediction.

To have the attributes for object oriented entities such as classes, we follow the nested approach as given above:

```
/* FooComment #class
id: 179
desc: [...]
*/
class className {
        [class implementation]

        /* FooComment #function
         [ ...Function parameters... ] */

        Function1 implementation

        /* end FooComment */

        /* FooComment #function
        [ ...Function parameters... ] */

        Function2 implementation

        /* end FooComment */
        ...
}
/* end FooCommment */
```

In the above example, we used nesting to catalog a class in C++. The outer comment defines the properties of the class, while the inner ones describe the functions. Whenever an */* end FooComment */* is encountered, it closes the innermost FooComment.

The inner blocks inherit the properties from the outer blocks and if there are conflicts, the inner block's attributes override the outer attributes within the scope of the block.

Given below is a small example implementation of the nested approach:

```
/* FooComment #class
id_keys: stack, LIFO
desc: implementation of stack capable of operating on
integer in C++
*/
class STACK {
        STACK() {
          top = -1;
        }
        ...
        /* FooComment #function
        i_arg: y
        o_arg: y
        inputs: integer[1]
        outputs: integer[1]
        id_keys: push
        desc: push integer to  the stack
        */
        int push(int n) {
          //check stack is full or not
          if(isFull()){
            return 0;
          }
          ++top;
          num[top]=n;
          return n;
        }
        /* end FooComment */
        ...
}
/* end FooComment */
```

Following the same approach, it is possible to create a catalog of all the information required for the bot to fulfil its purpose without interfering with the actual code or disrupting the flow of development.

III. DEVELOPMENT LIFE CYCLE

Development lifecycle of the Bot

After successfully cataloging the code, a bot can be created to be deployed on the software repository. The development lifecycle for the bot consists of the following steps:

1. Requirements gathering

Gather market requirements for the bot: The primary market for this implementation is IT and software industry. The design team should meet the developer teams in varied organizations and see the problems they face while development and what benefits the solution can deliver.

2. Specification

Develop a product spec for the bot identifying its features and functionality.

3. Script

While websites and apps have structured interfaces, the bots have a conversational interface. The conversational script must be precise and less time-consuming. The script may or may not support NLP capabilities. A conversation example is provided in the 'conceptual use' segment.

4. Development

Create the front-end and back-end components. The front-end refers to the conversational interface—translating user input into specific actions and vice versa. The back-end refers to cataloguing the existing code, database creation and integration with the database.

5. Training

If the bot has NLP or related ML capabilities, you need to train the bot to handle specific sets of conversational statements. Depending on the technology being used to run the bot engine, you may or may not require a training phase.

6. Testing

The working of the bot should be tested using real-world datasets for different programming languages.

7. Deploy

Once the bot is built, it must be deployed on the local server of the customer company or a public repository. Testing again may be required to see if the deployment is successful and if the bot is working as expected.

8. Monitor

Once the bot is deployed, it must be monitored. The bot should be monitored by reading conversational scripts flagged by the users in which the bot was unable to cater to the requirements. The approach will be effective to mark out the unresponsiveness of the bot towards certain user conversations.

9. Analyze

As the bot starts being used, its performance must be tracked, reports should be generated and the results should be analyzed.

Repeat

The learnings from the Analyze phase can be cycled back into the bot development process to build an ever-improving bot. Some bots may even be programmed with self-learning AI programs that need inputs from users and trainers to continuously improve themselves.

The complete lifecycle is shown in the Figure 1 given below:

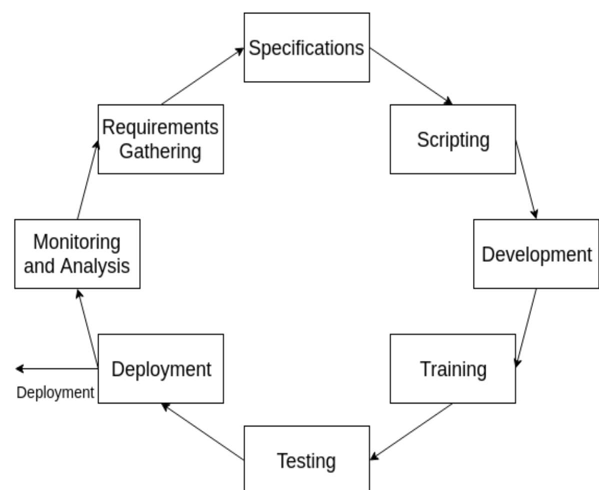

Fig. 1 Development life cycle of the Bot assisted Software Development System

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication

Workflow of the Customers

The workflow of the consumer organization and community doesn't need a huge adjustment and the companies can keep following the development cycle they are following prior to the deployment of the bot system.

The new systems that are being developed by the consumers of the bot, should include catalog comments as and when deemed necessary, during development itself. The only overhead is the preparation of the previously available code that requires manual comment addition during the initial setup of the system.

A Sample Use Case is provided in Appendix A.

IV. CONCLUSION

This paper identified the potential use of a bot to catalog and search the software database and to extract the code snippets from previous projects according to the requirement of the user. We saw the way to catalog the code to add enough information required to build a database of program and software source code.

Adding extra comments to the code will demand time but in turn, yield a clutter-free database that can be used to head start new projects without the need to go through the previous project to search reusable snippets. The comments can be removed afterwards from the final build in order to hide unnecessary information that may not be needed to ship out with the final system. A system like this has numerous advantages and the obvious achievable are:

- An Encyclopedia of Code Snippets, Functions and Programs.
- Retrieval of the Required Code Snippet without Manual Searching.
- Head Start in Any Project If The Required Functionality Is Found.
- Starting Anchor for New Projects and Inspiration for Adding New Functionalities.
- No Need to Code the General Structures Repeatedly.
- A Way to Catalog Legacy Code.

REFERENCES

1. Ahmed Mateen et al., A Software Reuse Approach and its Effects on Software Quality, an Empirical Study for Software Industry, 2017
2. G. Booch and J. Rumbaugh, The Unified Software Development Process, Addison-Wesley, 1999

AUTHOR INFORMATION

Ujjwel Balwal, Undergraduate student, Department of Computer Science and Engineering, SSN College of Engineering, Anna University, Tamil Nadu, India.
ujjwel17179@cse.ssn.edu.in

Dr. K. Vallidevi, Associate Professor, Department of Computer Science and Engineering, SSN College of Engineering, Anna University, Tamil Nadu, India.
vallidevik@ssn.edu.in

APPENDIX A

Use Case Example

The primary purpose of using such an assistant will be coming up with the primitive code, separated in modules collected from different parts of (mostly unrelated) programs. Although fragmented and not compatible with each other, this primitive code base can provide with the essential raw material and inspiration for the development of a complete program. The user (programmer) will provide with the requirements, the bot will search for related modules in the database and will arrange the code fragments as specified by the user. We would try to explain this using a painfully simple example. Assume the bot crawled our database and created the following dataset: (Only essential attributes are displayed due to space constraints)

i_arg?	outputs	o_arg?	p_lang	space_complexity	time_complexity	id_keys	desc	sp_dependencies
y	string[1]	n	py	n	n	url, text, website, parse	Saves a webpage as text [...]	urllib.request; text2html
n	string[1]; .txt	n	java	null	null	url, scraper	Extracts all the URLs from a webpage	beautifulsoup
y	int[1]	y	py	n	n^2	semantic analysis	[...]	nltk
y	string[1]; .csv	y	py	n	n	key value pair to csv	[...]	csv

Demo Script

Let's assume our bot's name is Neo;
Programmer needs to create a program to read some web pages and analyze them semantically.

```
NEO: Hello, Prompt?
PROGRAMMER: webpage semantic analysis
NEO: Found candidate(s)
            1) py - symanstic analysis: str[1] - int[1]
PROGRAMMER: add 1 to bag
//bag refers to our collection of code. By default 1 will be added to the end of the list of code blocks in the bag
NEO: Done. Prompt?
PROGRAMMER: website scraper
NEO: Found candidate(s)
            1) py - url to text: str[1] - str[n]; .txt
PROGRAMMER: show 1 detailed
//programmer did not know exactly which module the bot is talking about and thus prompts for a detailed description
NEO: Saves a webpage as text. A web scraper that converts webpage from the specified url to text file - input: string as argument - output: string without return in .txt - time complexity:O(n) - space complexity:O(n) - dependencies […]
PROGRAMMER: add 1 to bag[0]
//bag[0] means the starting of the list in the bag, before the previously added modules
NEO: Done. Prompt?
PROGRAMMER: show bag
NEO:
   1) website scraper
   2) semantic analysis
PROGRAMMER: checkout
```

Box. 1 Demo Conversational Script

The bot generates the raw code having the above modules in specified order

Although this may not be the exact way how the bot should behave and be used, implementations can differ; but at the end of the day, the developer should not worry about searching the previously available code, and his or her work should begin with customizing the snippets and modules, if available, instead of rewriting the code from the scratch.